Leçon d'informatique : Recherche du plus court chemin : heuristique A^*

S. Benlhajlahsen - PCSI$_1$

Sommaire

I Exemple : le problème du *pathfinding*

Les captures d'écran ci-dessous sont faites à partir du site https://qiao.github.io/PathFinding.js/visual/. Vous pouvez y préparer une carte avec des obstacles et regarder en animation toutes les cases analysées par divers algorithmes pour relier un point de départ à un point d'arrivée.
La figure 1 montre un exemple de problème de *pathfinding* : le point de départ est le vert (à gauche) et le point d'arrivée le rouge (à droite), et il y a des murs sur le chemin (en gris).

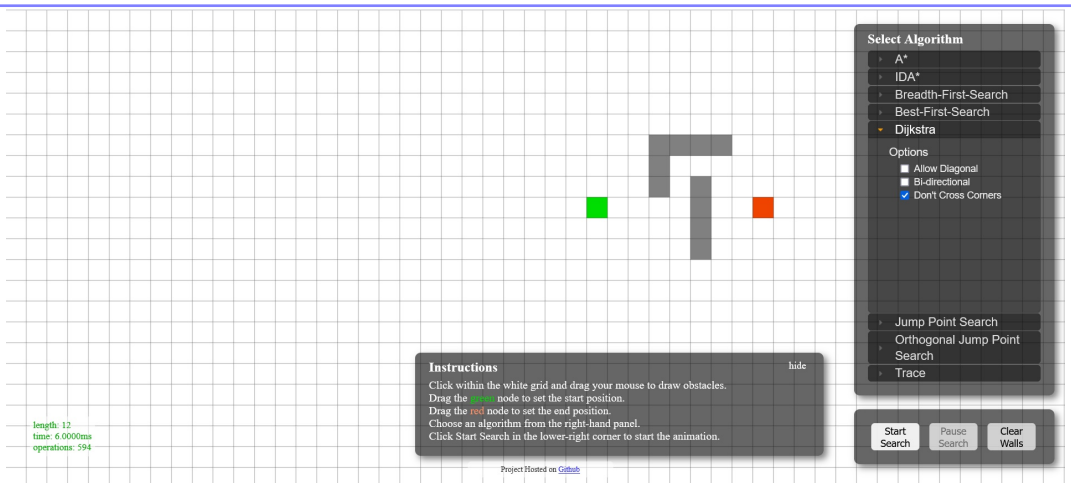


FIGURE 1 – Situation initiale du problème de *pathfinding*.

On associe à cette carte un graphe dont les sommets sont les cases accessibles (pas les murs), et deux sommets sont reliés par une arête si les deux cases sont adjacentes. On peut alors faire tourner l'algorithme de Dijkstra sur ce graphe pour trouver le plus court chemin entre les deux sommets choisis.
Même si on interrompt l'algorithme dès que le sommet d'arrivée est analysée, on aura parcouru beaucoup de cases, comme le montrent la figure 2.
Sans surprise, l'algorithme a essentiellement décrit des «cercles concentriques» autour du sommet de départ et donc parcouru des sommets qui, à l'œil nu, semblent clairement inutiles. En effet, il ne tient pas compte de la «direction» dans laquelle se trouve l'arrivée, et donc explore aveuglément dans toutes les directions.
On peut alors proposer de **biaiser** l'algorithme, en le modifiant pour qu'il explore en priorité les cases qui le **rapprochent** de la case d'arrivée. Il n'est pas garanti que ce soit une bonne idée : si par exemple il a un gros obstacle directement entre le départ et l'arrivée, un tel choix revient à aller vers le mur, alors qu'il aurait été plus malin de contourner.
Cette stratégie s'appelle A^* et est l'objet de ce chapitre. Le résultat du parcours par A^* est montré figure 3. Vous constatez que ce parcours se termine bien plus vite qu'avec Dijkstra, puisque beaucoup moins de cases sont considérées.
Ce qu'il faut retenir, c'est que c'est une stratégie **raisonnable**, mais pas forcément **optimale**. Il y a donc un compromis à trouver entre le risque de ne pas trouver le chemin le plus court, et l'économie de calcul.

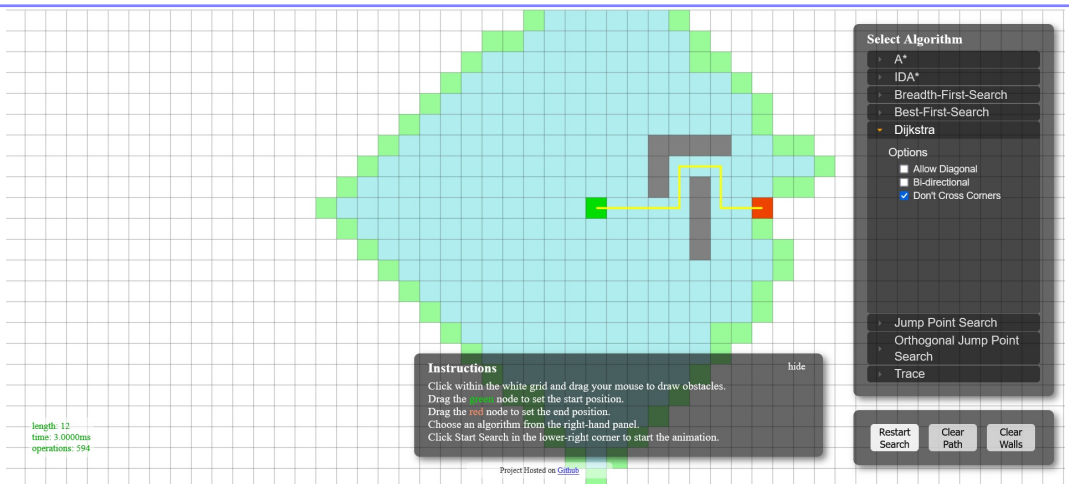


FIGURE 2 – Cases parcourues par l'algorithme de Dijkstra jusqu'à analyse du sommet d'arrivée.

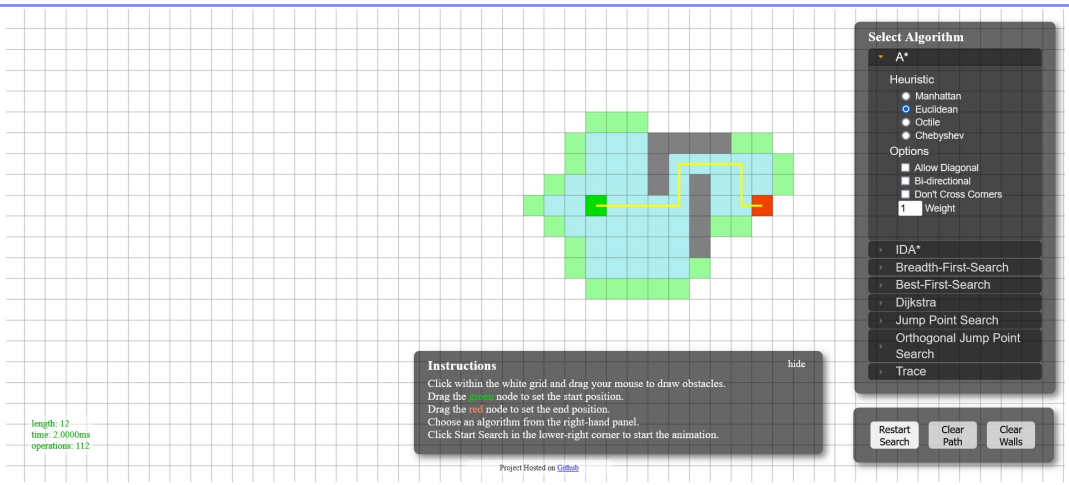


FIGURE 3 – Cases parcourues avec la stratégie A^*.

II Notion d'heuristique

> **À retenir :** Une heuristique est un processus qui s'apparente à un algorithme (traitement par itérations successives jusqu'à une condition d'arrêt) mais ne garantit pas de trouver la meilleure solution au problème posé. En contrepartie, elle doit converger vers une solution bien plus vite.
> En général, il n'y a aucun moyen rapide de savoir à quel point la solution renvoyée par l'heuristique est proche de la solution optimale. Il y a donc une part «artisanale» dans le choix et la conception des heuristiques.

A^* est une heuristique intéressante quand il est nécessaire de fournir une réponse **rapidement**. Par exemple, si vous utilisez le GPS de votre téléphone pour trouver votre chemin, vous accepterez de patienter quelques secondes, mais pas plusieurs heures ! Si le GPS utilise une heuristique qui conduit à un trajet qui n'est pas le plus court mais qu'il vous rallonge d'une durée faible devant la durée totale du trajet, elle sera considérée comme acceptable.

III Implémentation

A^* est en grande partie identique à l'algorithme de Dijkstra. On choisira la version de Dijkstra qui s'arrête dès qu'elle atteint le sommet désigné comme arrivée du parcours.

III.A Distance estimée passant par un sommet

Dans Dijkstra, un sommet était *push* dans la file de priorité avec la meilleure distance connue à ce stade entre lui et le départ.

À retenir : Dans A^*, on *push* le sommet avec une **estimation de la meilleure distance** entre le départ et l'arrivée passant par ce sommet.

Pour cela, avant de lancer l'heuristique, il faut fournir une donnée supplémentaire avec le graphe : la **distance à vol d'oiseau de chaque sommet à l'arrivée.** Comme son nom l'indique, cette distance ne correspond pas forcément à celle que l'on parcourt vraiment.

- Si le graphe représente une carte géographique, on peut littéralement prendre la distance à vol d'oiseau entre les sommets (typiquement, avec les coordonnées des points de départ et d'arrivée, on calcule cette distance par théorème de Pythagore).
- Sinon, il faut se donner un critère d'évaluation plus ou moins arbitraire, en fonction du problème posé.

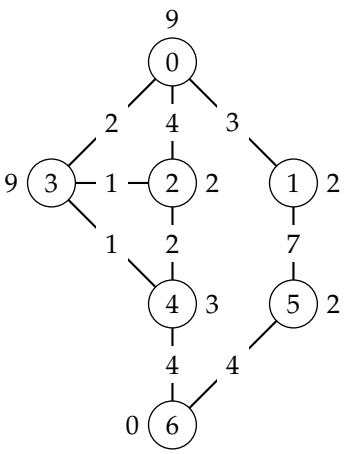
Soit d le sommet de départ et a le sommet d'arrivée, et u un sommet quelconque.

- On note $\text{dist}(u)$ la meilleure distance connue à ce stade de d à u, comme pour Dijkstra. On rappelle qu'elle peut varier pendant l'exécution.
- On note $\text{oiseau}(u)$ la distance à vol d'oiseau estimée de u à a, fixée au départ.
- La meilleure distance estimée de de d à a passant par u est alors $\text{dist}(u) + \text{oiseau}(u)$.

III.B Exécution manuelle sur un exemple

On prend le graphe ci-dessous et on choisit comme départ le sommet 0 et comme arrivée le sommet 6.

Initialisation :



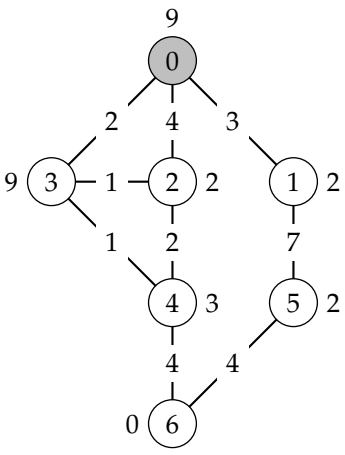
Les distances à vol d'oiseau de l'arrivée sont écrites à côté des sommets sur la figure, et reportées dans le tableau ci-dessous.
Les valeurs initiales des distances au sommet de départ et des prédécesseurs sont identiques à Dijkstra.
On *push* le sommet de départ accompagné de la distance estimée à l'arrivée $0 + 9 = 9$.

Sommet	0	1	2	3	4	5	6
Distance	0	∞	∞	∞	∞	∞	∞
Prédécesseur	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
Vol d'oiseau	9	2	2	9	3	2	0

File de priorité (0,9)

Comme c'est le sommet 0 qui sort quand on *pop* la file de priorité, on commence la boucle sur les voisins de 0. Il va y avoir trois tours.

Premier tour :



On commence par le sommet 1. Il n'est pas marqué, donc :

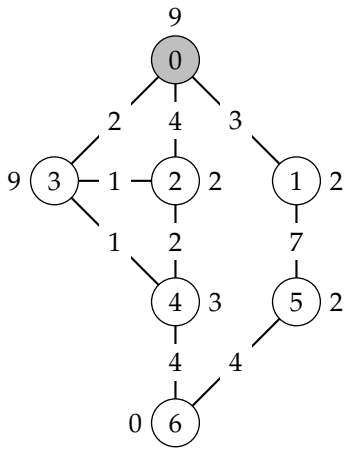
- Il n'a pas de prédécesseur, que l'on met donc à 0.
- Sa meilleure distance connue au départ est 3.
- La meilleure estimation de distance de 0 à 6 passant par 1 est $3 + 2 = 5$.

Puis on le *push*.

Sommet	0	1	2	3	4	5	6
Distance	0	3	∞	∞	∞	∞	∞
Prédécesseur	$\emptyset$	0	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
Vol d'oiseau	9	2	2	9	3	2	0

File de priorité : (1,5)

Premier tour (suite) :



Le sommet 2 n'a pas encore de prédécesseur, donc on le met à 0, meilleure distance connue au départ 4, meilleure estimation de distance de 0 à 6 passant par $2 : 4 + 2 = 6$. Puis on le *push*.

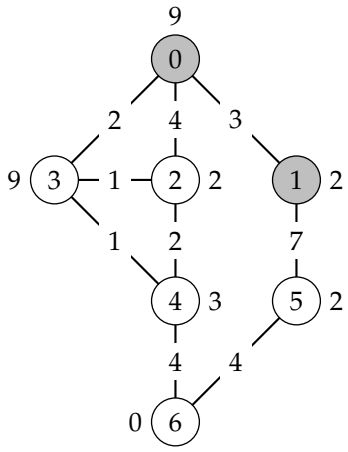
Le sommet 3 n'a pas encore de prédécesseur, donc on le met à 0, meilleure distance connue au départ 2, meilleure estimation de distance de 0 à 6 passant par $3 : 2 + 9 = 11$. Puis on le *push*.

Sommet	0	1	2	3	4	5	6
Distance	0	3	4	2	∞	∞	∞
Prédécesseur	$\emptyset$	0	0	0	$\emptyset$	$\emptyset$	$\emptyset$
Vol d'oiseau	9	2	2	9	3	2	0

File de priorité : (1,5)(2,6)(3,11)

La boucle sur les voisins de 0 est finie. Donc on *pop* la file de priorité, ce qui fait sortir le sommet 1, qui est associé à la meilleure distance actuellement connue de 0 à 6 (de longueur 5). La boucle sur les voisins de 1 commence et va faire un seul tour puisque 0 est déjà marqué.

Deuxième tour :



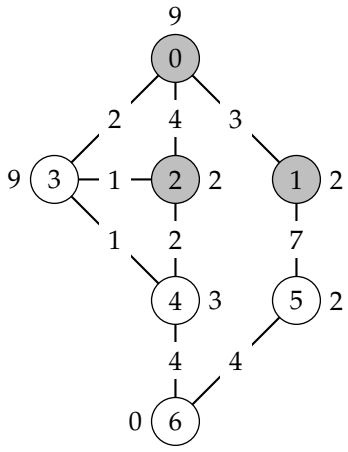
Sommet 5 sans prédécesseur, que l'on met à 1, meilleure distance connue au départ $3 + 7 = 10$, meilleure estimation de distance de 0 à 6 passant par $5 : 10 + 2 = 12$. Puis on le *push*.

Sommet	0	1	2	3	4	5	6
Distance	0	3	4	2	∞	10	∞
Prédécesseur	$\emptyset$	0	0	0	$\emptyset$	1	$\emptyset$
Vol d'oiseau	9	2	2	9	3	2	0

File de priorité : (2,6)(3,11)(5,12)

Le prochain *pop* fait sortir le sommet 2, qui a deux voisins non marqués.

Troisième tour :



Sommet 3 : il a déjà un prédécesseur. Sa distance à 0 venant de 2 est $4 + 1 = 5$, ce qui est moins bien que la distance déjà connue (2), donc on ne touche à rien.

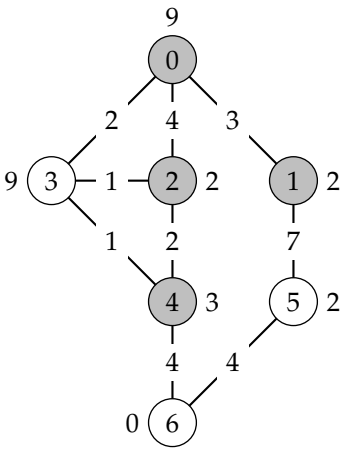
Sommet 4 : pas encore de prédécesseur donc on le met à 2, meilleure distance connue au départ $4 + 2 = 6$, meilleure estimation de distance de 0 à 6 passant par $4 : 6 + 3 = 9$.

Sommet	0	1	2	3	4	5	6
Distance	0	3	4	2	6	10	∞
Prédécesseur	$\emptyset$	0	0	0	2	1	$\emptyset$
Vol d'oiseau	9	2	2	9	3	2	0

File de priorité : (3,11)(5,12)(4,9)

Le prochain *pop* fait sortir le sommet 4, qui a deux voisins non marqués.

Troisième tour (suite) :



Sommet 3 : il a déjà un prédécesseur. Sa distance à 0 venant de 4 est $6+1=7$, ce qui est moins bien que la distance déjà connue (2), donc on ne touche à rien.

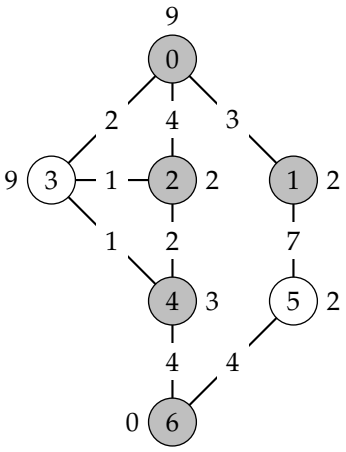
Sommet 6 : pas encore de prédécesseur, on le met à 4, meilleure distance connue au départ $6+4=10$, meilleure estimation de distance de 0 à 6 passant par 6 : $6+3=9$. Puis on le *push*.

Sommet	0	1	2	3	4	5	6
Distance	0	3	4	2	6	10	10
Prédécesseur	∅	0	0	0	2	1	4
Vol d'oiseau	9	2	2	9	3	2	0

File de priorité : (3,11)(5,12)(6,9)

Le prochain *pop* fait sortir le sommet 6. Il a un voisin non marqué.

Fin :



Sommet 5 : il a déjà un prédécesseur. Sa distance à 0 venant de 6 est $10+4=14$, ce qui est moins bien que la distance déjà connue (10), donc on ne touche à rien.

Sommet	0	1	2	3	4	5	6
Distance	0	3	4	2	6	10	10
Prédécesseur	∅	0	0	0	2	1	4
Vol d'oiseau	9	2	2	9	3	2	0

File de priorité : (3,11)(5,12)

Comme 6 est le sommet d'arrivée et qu'il est maintenant marqué, le processus s'arrête.

Conclusion : L'heuristique A^* propose donc un chemin de longueur 10 du sommet 0 au sommet 6. On reconstitue ce chemin avec les prédécesseurs :

- 6 a pour prédécesseur 4.
- 4 a pour prédécesseur 2.
- 2 a pour prédécesseur 0.

Donc le chemin est $0 \to 2 \to 4 \to 6$.
Sur ce petit graphe, vous pouvez voir à l'œil nu que ce n'est pas le plus court chemin : $0 \to 3 \to 4 \to 6$ est de longueur 7. Ceci montre que A^* n'est pas un algorithme qui détermine le plus court chemin. Mais il a proposé un chemin relativement court, et il n'a pas eu besoin d'explorer tous les sommets pour ça et donnera donc sa conclusion plus vite que Dijkstra.
L'interprétation est simple : la distance à vol d'oiseau du sommet 3 à l'arrivée a été évaluée à 9, ce qui est très important et a «découragé» A^* de considérer les chemins passant par 3. Si c'est une carte géographique, cela revient à considérer que vous avez fortement surestimé la distance de 3 à 6, ce qui a faussé votre conclusion.

III.C Code

On reprend la fonction `dijkstra` du chapitre dans sa version où la boucle `while` s'arrête dès que le sommet d'arrivé est marqué.

- On peut réutiliser la file de priorité.
- On peut réutiliser la fonction `init_dijkstra` pour générer les variables de travail.

- On peut réutiliser les fonctions `est_tendu` et `relaxation` sans modification.

```
def est_tendu(depart, arrivée, poids, distance):
    return distance[depart]+poids < distance[arrivée]
def relaxation(depart, arrivée, poids, distance, prédécesseur):
    distance[arrivée] = distance[depart]+poids
    prédécesseur[arrivée] = depart
```

```
1  def Aetoile(graphe, depart, arrivée, oiseau):
2      distance, prédécesseur, marqué, fp = init_dijkstra(graphe)
3      distance[depart] = 0
4      file_prio_push(fp, [depart, oiseau[depart]])
5      while not file_prio_vide(fp) and not marqué[arrivée]:
6          sommet, _ = file_prio_pop(fp)
7          if not marqué[sommet]:
8              marqué[sommet] = True
9              for voisin, poids_voisin in graphe[sommet]:
10                 if est_tendu(sommet, voisin, poids_voisin, distance):
11                     relaxation(sommet, voisin, poids_voisin, distance, prédécesseur)
12                     file_prio_push(fp, [voisin, distance[voisin]+oiseau[voisin]])
13     return distance, prédécesseur
```

Il n'y a finalement que trois lignes à changer :

- la ligne 1, afin de passer la **liste** des distances à vol d'oiseau à la fonction,
- les lignes 4 et 12, pour que les *push* se fassent avec la meilleure estimation du départ à l'arrivée passant par le sommet.

Remarquez bien une chose : Pour déterminer si un arc est tendu, on reste sur l'utilisation de la meilleure distance connue au sommet de départ, comme avec Dijkstra.
La meilleure distance estimée du départ à l'arrivée passant par un sommet n'est **pas** utilisée en dehors du *push* (et donc du *pop*).